Efficient Disaggregated Memory Eviction with Glitter

Linxuan Zhong, Wenxin Li, Yulong Li, Jiawen Shen, Song Zhang, Wenyu Qu, Yitao Hu
College of Intelligence and Computing, Tianjin University, China
Email: {sylas, toliwenxin, toliyulong, shenjiawen, zhang_song, wenyu.qu, yitao}@tju.edu.cn

***Abstract*—Memory disaggregation, a promising technique allowing applications to use remote memory, is increasingly appealing in datacenters due to its high resource utilization. Operationally, the application's host server constantly evicts unused data to remote to make room for memory allocation of new pages. Inefficient evictions allow memory usage to hit its limit, resulting in application blocking, which brings severe throughput degradation. However, most existing works neglect the importance of eviction. They offload the eviction to a background thread and set a fixed trigger timing, rendering a belated eviction. Worse still, they overlook the impact of network congestion on eviction efficiency, making their strategy flawed in large-scale scenarios. In this paper, we present Glitter, an adaptive, multi-level awareness eviction solution that accelerates applications by minimizing the overhead of application blocking from host and network aspects. For host, Glitter presents an adaptive eviction threshold adjustment to optimize the eviction timing, reducing the occurrence of application blocking. For network, Glitter adopts an eviction flow scheduling to address the hazards posed by flow contention at switches, decreasing the duration of each application blocking. Through comprehensive experiments, Glitter gives an average 1.4× throughput boost to Fastswap, a state-of-the-art disaggregated memory system.**

***Index Terms*—Resource disaggregation, remote memory, datacenter networks**

I. Introduction

Memory disaggregation, a promising technique that enables applications to access memory on remote servers, has gained traction due to its cost-efficiency and flexibility [1]–[12]. Traditionally, a memory disaggregation system consists of compute and memory servers connected through a remote direct memory access (RDMA) network. The compute server running the application is typically equipped with a small portion of local memory. Therefore, the compute server typically needs to (1) evict unused data from local to remote memory and (2) fetch demand data from remote to local memory.

Most existing works [1], [2], [10], [11], [13] focus on prefetching to hide data fetching latency but overlook the importance of data eviction. These systems often maintain an asynchronous background thread to perform data eviction. Operationally, this thread monitors local memory usage and executes eviction when memory consumption exceeds a critical watermark or hits its limit. They typically set a high threshold to maximize memory utilization. However, this fixed threshold may lead to inopportune data eviction (§ II-C). Even worse, they ignore the impact of network congestion on eviction capacity in large-scale scenarios and count on eviction being stable and sufficient to meet the memory demands [4], [5]. Experimental evidence indicates the naivety of this assumption. For example, pagerank has experienced 14.7× eviction performance degradation due to network congestion (§ II-C). When the memory growth rate exceeds the eviction rate, applications have to wait for memory eviction to free up enough space for demand data. We show that kmeans spends 58% of time waiting for data eviction (§ II-B). Therefore, efficient memory eviction is crucial to avoid application blocking.

Actually, the overhead of application blocking is affected by both *host* and *network* aspects. On the one hand, the host side determines the timing of starting data eviction. Timely data eviction ensures the availability of local memory, mitigating potential occurrences of application blocking in the future. For network, the round-trip time (RTT) of evicted data flow directly impacts the duration of each application blocking event. The memory disaggregation architecture breaks down conventional monolithic servers into numerous compute and memory nodes. This proliferation of nodes places significant pressure on the network, inevitably causing evicted data flows to queue up at switches. Hence, extended RTT reduces the efficiency of data eviction, causing the local memory not to be freed up in time and further exacerbating the negative impact of application blocking. Therefore, alleviating or even eliminating the impact of application blocking requires joint efforts from both the host and network aspects.

In this paper, we propose Glitter, an *adaptive, multi-level awareness eviction solution* that minimizes the overhead of application blocking for memory disaggregated applications. Unlike existing eviction solutions that simply asynchronously offload evictions to a background thread or dedicated CPU, Glitter not only determines the appropriate timing of eviction from the host side but also mitigates the impact of congestion on eviction from the network side based on multiple awareness mechanisms. Precisely, Glitter consists of two modules that reduce the overall application blocking overhead by reducing the number of blocking occurrences and the duration of each blocking event to improve application throughput.

- ***Adaptive Eviction Threshold Adjustment Module.*** In this module, we design an early eviction mechanism to predict the potential for application blocking based on *memory growth awareness* and *network congestion awareness* during application runtime. By opportunistically lowering the

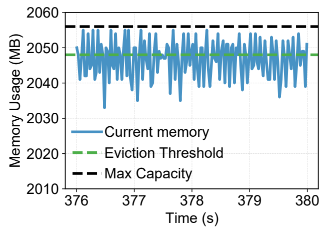


Fig. 1: The memory usage of a Java program.

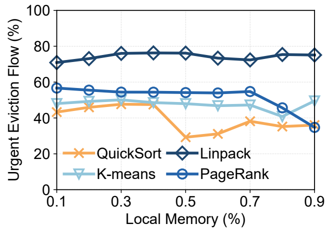


Fig. 2: The ratio of UEFs to total eviction flows.

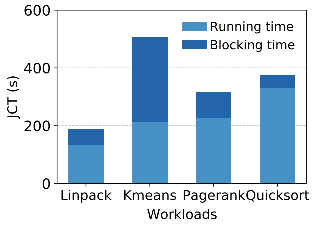


Fig. 3: The JCT breakdown under 50% local memory.

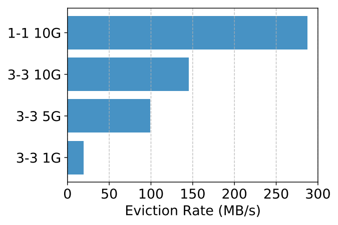


Fig. 4: App perf. degradation under network congestion.

eviction threshold, data eviction can be triggered early. Meanwhile, to avoid premature eviction causing memory underutilization, we present a pullback mechanism under the guidance of *memory utilization awareness* to avoid excessive eviction threshold reduction. In this way, we can free up memory promptly to mitigate or even eliminate potential memory exhaustion, thereby reducing the number of application blocking occurrences.

- ***Eviction Flow Scheduling Module.*** Since network congestion leads to longer waiting times for blocked applications, we propose a novel flow scheduling strategy to eliminate unfair flow contention at the switch. Concretely, in compute servers, we assess the impact of different flows on application blocking and conduct an urgency-level analysis of evicted data flow at the switch, prioritizing flow with higher urgency. The urgency-level is determined based on *memory exhaustion awareness*. We employ the weighted round-robin (WRR) scheduling algorithm at the switches to prevent starvation of low-priority eviction flows.

We incorporate Glitter into Fastswap [1], a state-of-the-art kernel-integrated disaggregated memory system, and evaluate the performance compared to the original version through comprehensive experiments with four data analysis workloads under varying local memory ratios. In end-to-end performance evaluation, we find that Fastswap with Glitter always achieves better performance and reaches an average 1.4× throughput performance improvement compared to the original Fastswap.

II. Background and Motivation

A. Preliminaries

Memory Disaggregation: Traditional server-centric architecture arranges the CPU and memory within a single monolithic server. Unfortunately, memory-intensive applications (e.g., data analytics and ML workloads [14]–[17]), which hold substantial volumes of intermediate data in memory for quick processing, are facing memory capacity walls due to the poor scalability and low memory utilization inherent in this architecture [1], [18]. Memory disaggregation, an effective solution for addressing memory bottlenecks, is increasingly become attractive recently [1]–[11]. Memory disaggregation systems break the monolithic servers into independent compute and memory nodes, where compute nodes typically possess KB/MB local cache memory. In contrast, memory nodes have GB/TB remote memory but limited computation capability (e.g., only 1-2 CPU cores) [19]. These two components communicate via high-speed remote direct memory access (RDMA) networking.

Data Fetching and Eviction: Memory disaggregation systems perform two primary operations: data fetching and data eviction. For the former, when an application accesses to non-local data, it necessitates its retrieval from a remote server. Most implementations (e.g., [1], [4]–[6], [10], [20]) achieve data fetching by modifying page fault handler, which employs RDMA `read` as the communication primitive. For the latter, the compute nodes must evict locally unused data to remote memory by issuing RDMA `write` when the local memory is running low. Existing systems [1], [2], [20] use a dedicated eviction thread or CPU to execute data eviction. During the application runtime, the eviction thread continuously monitors the memory usage. Once it exceeds the predefined eviction threshold, the eviction thread will initiate the data eviction until it again falls below the eviction threshold. We define the flow from transferring unused data to remote memory as eviction flow. We classify eviction flow into two types based on the states of the application. *Regular eviction flow (REF)* is generated by an application whose current memory usage exceeds the eviction threshold but remains below the maximum memory capacity. At this point, the application still has available local memory. *Urgent eviction flow (UEF)* is generated by an application whose remaining amount of memory is approaching its maximum capacity and cannot accommodate the required data anymore.

B. Application Blocking Due to Urgent Eviction

Problem Statement: We argue that application blocking occurs when the system approaches its maximum memory capacity and generates UEF. During the application runtime, if the memory growth rate exceeds the eviction rate of the eviction thread, the local memory usage will surpass the eviction threshold and keep rising until it reaches maximum memory capacity, leading to memory exhaustion. At this point, if the application needs to allocate pages, the CPU executing the application will be scheduled to perform data eviction in conjunction with the eviction thread. Then, the local host will wait for an acknowledgment message from the remote server to confirm the successful eviction. In this process, the application becomes blocked and cannot proceed until enough memory space is liberated. Therefore,

TABLE I: Application performance degradation due to flow contention and congestion at the switches.

Workloads	*Standalone JCT (s)*		*Multi. JCT (s)*		*Slowdown for*
	User Time	*Sys. Time*	*User Time*	*Sys. Time*	*Sys. Time*
linpack	93.28	**242.20**	98.76	**484.80**	2.00
kmeans	217.92	**1076.56**	217.22	**1355.01**	1.26
pagerank	168.34	**556.66**	168.32	**891.12**	1.60
quicksort	239.99	**121.50**	240.63	**147.03**	1.21

UEF's flow completion time (FCT) will have a direct impact on the duration of application blocking. Subsequently, the application CPU resumes the execution of the application process, which will lead to memory exhaustion again. The cycle repeats as shown in Fig. 1, resulting in a significant number of application blocking.

Problem Analysis: Application blocking slows down applications' completion time, especially for memory-intensive applications that frequently communicate with memory. We conduct an experimental analysis to examine this issue. We run our experiments on two directly connected `xl170` machines on CloudLab [21]. One machine acts as the compute node, and the other as the memory node. We utilized Fastswap [1] as the testbed and employed four workloads provided by the CFM [22] framework. The amount of memory used and the number of CPU cores follow CFM's default configuration.

As shown in Fig. 2, even with 90% local memory of the entire working set size, the eviction flows generated by the blocked application (i.e. UEFs) still comprise an average of 49% of the total, while the REFs account for 51% accordingly. Fig. 3 shows the impact of application blocking on the throughput of different applications. On average, approximately 32% of time is wasted on blocking waits for these four types of applications. As for kmeans, an iterative clustering algorithm that requires extensive memory access, it has a blocking time proportion of up to 58%.

Existing Works: Most existing works [1], [2], [4], [20] asynchronously offload eviction to dedicated threads or CPUs, which are assumed to have the capability to deal with the growing memory demands of applications, to avoid data eviction on the critical path when accessing data. Infiniswap [6] employs a daemon to periodically evict data when memory usage exceeds the eviction threshold. Fastswap [1] proposes to asynchronously offload the eviction process to the dedicated CPU core. AIFM [2] assigns higher priority to eviction threads, ensuring more immediate attention to data eviction tasks. DiLOS [5] attempts to always keep a few free pages by eagerly evicting through a cleaner and a reclaimer. Hermit [4] adaptively increases the number of eviction CPUs when the eviction rate is lower than the memory growth rate.

C. Limitations of Existing Works

Fixed Eviction Threshold Leads to Performance Tradeoff. Most existing memory disaggregation systems use a fixed eviction threshold [1]–[3], [6], [9], [10], [20]. For example, in Fastswap [1], the threshold is set to the maximum memory capacity minus 8MB. However, this approach of manually and statically setting the eviction threshold based on experience makes the triggering of data eviction inflexible. A high eviction threshold leads to belated eviction when faced with prone-to-blocked applications, which potentially run out of memory in future. While a low eviction threshold results in premature eviction, reducing memory utilization for non-blocked applications. Underutilized local memory causes frequent communication between local and remote memory, degrading application throughput.

Specifically, we designed two Java applications to illustrate this issue. They continuously create objects, causing memory usage to increase at a specific rate. One of the Java applications has a faster memory growth rate and experiences frequent application blocking, while the other does not. We find that lowering the threshold by 80% for faster eviction in blocked Java application in Fig. 1 leads to a decrease of approximately 53% in the number of UEFs, as well as a 10% reduction in application completion time. In contrast, for the non-blocked Java application, this results in a 4.96$\times$ completion time increase with the low memory utilization.

Network Congestion Leads to Slower Eviction. In memory disaggregation systems, the original data communication between CPU and memory has shifted from PCIe to the network, which becomes a critical bottleneck for application performance [18]. To observe the impact of network congestion, we expand the topology to a three-to-three configuration, where three computer nodes and three memory nodes are connected through two layers of three switches. We measured the results in a one-to-one topology as a reference. Each link has a bandwidth of 10GB. The results in Table I shows that network congestion slows down system time by 1.2$\times$~2.0$\times$.

Network congestion degrades performance by reducing the efficiency of data eviction. The queuing delay at switches, caused by many eviction flows competing, extends UEFs' FCT and causes longer waits for applications. To explore the effect of network congestion on eviction rate, we conduct tests under four network scenarios, including one-to-one with 10G bandwidth and three-to-three with {10G, 5G, 1G} bandwidth. The results are measured for pagerank with 25% local memory. As shown in Fig. 4, when network congestion intensifies, the eviction rate drops significantly, by as much as 14.7$\times$.

However, most existing works ignore the impact of the network congestion, simplifying the scenario to small scale [7], [13] even two directly connected servers [2], [4], [5], [9], [10], where no network congestion exists. This neglect makes their solutions seriously deficient in dealing with large-scale scenarios. The eviction CPU cores allocation strategy in Hermit [4] becomes ineffective under fluctuating eviction rates. DiLOS [5] cannot assume a sufficient eviction rate to ensure enough memory is always available locally. Application blocking cannot be avoided through existing approaches.

D. Intuition and Challenges

Intuition: Considering the total time of application blocking is the cumulative product of the number of blocking

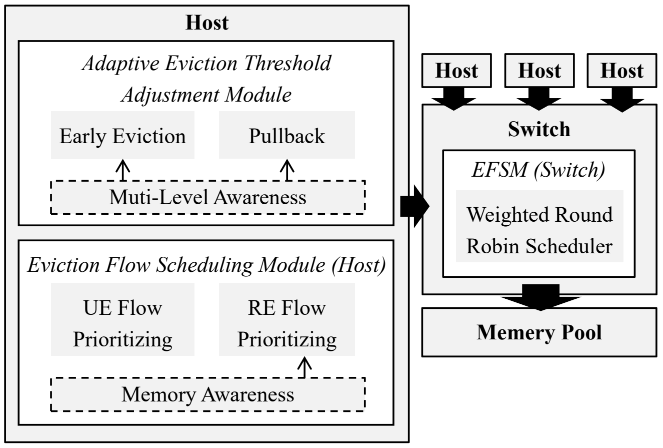


Fig. 5: Glitter overview.

occurrences and the duration of each blocking event, we give the following two intuitions to alleviate application blocking. (1) In contrast to existing systems that statically set eviction thresholds, we propose an *adaptive eviction threshold adjustment* strategy to reduce the number of application blocking occurrences. through which systems can optimize eviction timing and perform data evictions earlier before memory exhaustion, freeing up memory in advance to mitigate or even eliminate the occurrence of application blocking. (2) For network, we propose an *efficient eviction flow scheduling* strategy to mitigate the impact of flow contention at switches on the application's duration of each blocking event. To this end, we analyzed which eviction flows most affect application blocking and prioritized more urgent ones to lessen the network congestion's impact on application performance.

Challenges: Glitter overcomes two challenges. (1) How can we sense potential application blocking to lower the eviction threshold while avoiding memory underutilization? Glitter adjusts the eviction threshold by considering both memory and network. First, we predict the likelihood of application blocking occurring based on both the application's memory growth and the system's eviction rate to lower the threshold correspondingly. Then, we pull back the threshold when memory utilization is affected. (2) How can we sense the urgency level of eviction flow for prioritization? We ensure the strategy is widely applicable by focusing on scenarios using only commercial switches, requiring the host side to handle urgency-aware operations. Intuitively, UEFs get highest priority in scheduling as they directly influence application block times. For REFs, their urgency levels should be determined by both the memory demands of applications and the amount of remaining memory. However, when assigning priorities, the urgency levels of other REFs also needs to be considered, which are agnostic to the current computer node. Although information synchronization can be achieved through a central controller, it introduces additional overhead. To overcome this, we propose to estimate the urgency level of the current REF in the entire cluster based on historical data.

III. Glitter Design

A. Overview

Glitter aims to provide an efficient eviction solution to minimize overhead from application blocking, improving throughput for applications running in the memory disaggregation system. Compared with current data eviction solutions, Glitter has two following features: (a) For the host side, Glitter designs an eviction threshold adjustment mechanism based on a multi-level awareness approach, which enables the system to adaptively control the timing of data eviction (b) For the network side, Glitter proposes a novel flow scheduling strategy based on a memory awareness approach to mitigate the unfair competition among eviction flows at switches Fig. 5 provides an overview of Glitter, which mainly consists of two modules: an adaptive eviction threshold adjustment module and an eviction flow scheduling module, corresponding to the features depicted in (a) and (b), respectively.

B. Adaptive Eviction Threshold Adjustment Module

Glitter's A-ETAM aims to reduce the number of application blocking occurrences by predicting potential memory exhaustion and performing early data eviction to free up available memory. Based on a multi-level awareness mechanism, Glitter adaptively lowers the eviction threshold to trigger early eviction and mitigates the memory utilization decrease caused by premature eviction through a pullback mechanism.

Early Eviction with Two-Level Awareness: Our two-level awareness mechanism is built upon the fact that application blocking occurs when application's memory growth rate surpasses the eviction rate of the eviction CPU. Thus, by monitoring the relationship between V_{app} and V_{evict}, we can predict future blocking and lower the eviction threshold for early eviction. V_{app} and V_{evict} are calculated as follows.

1) Calculate V_{app} (Memory Growth Awareness): Calculating the memory growth rate requires obtaining the change in memory usage over a specific period as well as the elapsed time. To minimize CPU overhead, we specifically avoid implementing a timer-like design, which requires running in a separate thread and performing periodic time checks to trigger corresponding operations. We observe that in the Linux kernel, memory allocation and accounting are monitored by the `try_charge()` function in the memory management module, which provides an opportunity for our calculation. Therefore, Glitter records the change in memory usage and the elapsed time during N cycles when `try_charge()` is called in order to calculate the current memory growth rate.

2) Calculate V_{evict} (Network Congestion Awareness): In large-scale scenarios, the eviction rate becomes unstable due to the impact of network congestion. To calculate it, Glitter employs a counter that increments every time data is evicted, similarly using N `try_charge()` calls as a calculation cycle. For page-based systems, the granularity of data evicted is fixed (typically 4KB), making it easy to calculate the total amount of evicted data in a cycle. By dividing by the elapsed time, the eviction rate can be calculated.

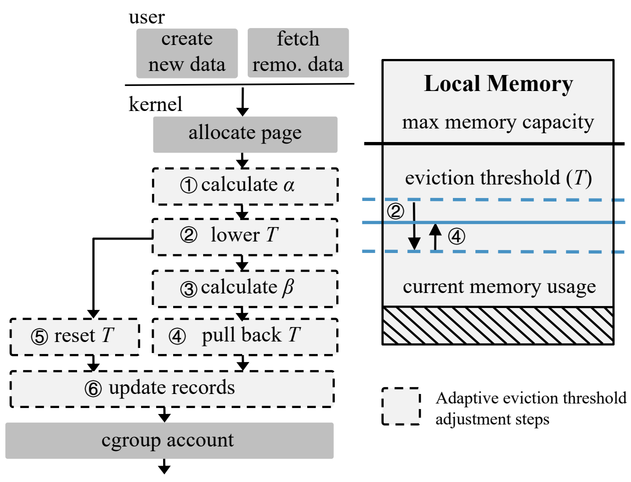


Fig. 6: A-ETAM workflow

Pullback with Memory Utilization Awareness: Lowering the eviction threshold involves a trade-off between reducing the number of blocking occurrences (which improves performance) and decreasing memory utilization (which impairs performance), inspiring Glitter to take memory utilization into account and propose a pullback mechanism. However, directly obtaining memory utilization is challenging due to frequent memory allocation and release. To tackle this problem, we define the *cache miss rate* ($CM_{current}$) as the ratio of non-local data accesses (N_{miss}) to the total number of accesses (N_{access}) within a cycle, as an indirect metric of memory utilization. An higher $CM_{current}$ suggests a lower memory utilization, indicates threshold need to be pulled back.

Workflow of Adaptive Eviction Threshold Adjustment: Fig. 6 shows the workflow of A-ETAM, which includes six steps. The details of each are described below.

1) Calculate α: The eviction threshold is initially set to T_{upper}. In order to quantify the relationship between memory growth rate and eviction rate, we calculate the following ratio:

$$\alpha = \frac{V_{app}}{V_{evict}} \tag{1}$$

2) Lower Eviction Threshold: $\alpha > 1$ means the eviction rate cannot keep up with the memory growth rate after the memory usage reaches the eviction threshold. To ease potential memory pressure, lowering the eviction threshold to start data eviction earlier and free up memory proactively becomes imperative. The value of α reflects the extent of the disparity between the application's memory demands and the eviction capabilities. A higher α indicates a higher risk of future memory exhaustion and application blocking, requiring a lower eviction threshold for prompt data eviction. Thus, we give the formula (Eq. 2) to update the eviction threshold. Note that the eviction threshold is constrained to be above T_{lower} to avoid over-lowering.

$$T'_{update} \leftarrow max\{\frac{T_{current}}{\alpha}, T_{lower}\} \tag{2}$$

Otherwise $\alpha \leq 1$, it will jump to **Step 5**).

3) Calculate β: To quantify the change in cache miss rate over a cycle, we consider the following ratio:

$$\beta = \frac{CM_{current}}{CM_{last}} \tag{3}$$

$\beta \leq 1$ indicates that the adjustment of the eviction threshold in the previous cycle did not have a negative impact on the local memory utilization rate. Therefore, the value of the eviction threshold will be maintained without any pullback.

4) Pull Back Eviction Threshold: When $\beta > 1$, it implies that memory utilization is affected by the lowering of the eviction threshold, which consequently necessitates a pullback. To prevent excessive pullback, we provide the following adjustment formula (Eq. 2) to constrain the additional updating of T_{update} from exceeding its original value $T_{current}$ or falling below the minimum threshold T_{lower}.

$$T_{update} \leftarrow T'_{update} * \frac{1}{\frac{1-\alpha}{\beta} + \alpha} \tag{4}$$

Due to the range of values for β being $(1, +\infty)$, the value of T_{update} will be constrained within $(T'_{update}, T_{current})$. In the worst case, T_{update} can almost be pulled back to the value at the time before lowering the eviction threshold, meaning that in this cycle Glitter will reject a decrease in the threshold.

5) Reset Eviction Threshold: $\alpha \leq 1$ indicates that the current eviction capability of the eviction CPU can handle the growth of application memory. Once the memory usage reaches the eviction threshold, it will no longer continue to increase without any concerns about future blocking. Therefore, the eviction threshold can be reset back to its original maximum value T_{upper} without the need for early eviction.

Given the fluctuating network and demands of applications, this situation may be temporary. Simply resetting the eviction threshold to T_{upper} is too aggressive and could negate the early eviction strategy. Cautiously, we introduce a conservative reset approach, where $T_{current}$ is incrementally reset each cycle, gradually approaching T_{upper} over M rounds of cycles.

6) Update Records: Finally, as a prelude to the subsequent cycle of calculations, Glitter proceeds to update several parameter values, including timestamps, the last memory growth rate, the last cache miss rate, etc.

C. Eviction Flow Scheduling Module

In large-scale memory disaggregation systems, the competition among eviction flows at switches has a detrimental impact on the eviction rate of blocked applications, resulting in a decrease in their throughput. To alleviate this problem, Glitter puts forth a novel and efficient memory-aware eviction flow scheduling module (EFSM) to enhance the eviction efficiency of both blocked and prone-to-blocked applications, thereby boosting their overall throughput. Glitter's EFSM primarily comprises two operations: prioritizing and scheduling.

UEF Prioritizing: In the EFSM, the highest priority is assigned to UEFs because UEFs' FCT directly impacts application throughput by prolonging the duration of application

Algorithm 1: Eviction Flow Prioritization Assignment

```
Input: Current memory remaining $M_{remain}$, Total memory $M_{total}$, Priority rule map $Map_p$, Current memory growth rate $V_{app}$
Output: Priority of the current eviction flow $P_e$
1  if Application is blocked then
2      $P_e \leftarrow P_1$
3  else
4      if $Map_p \neq NULL$ then
5          $T_{exhausted} \leftarrow M_{remain}/V_{app}$
6          $P_e \leftarrow$ Look up $Map_p$ with $T_{exhausted}$
7      else
8          $R_{remain} \leftarrow M_{remain}/M_{total}$
9          $P_e \leftarrow$ Calculate priority with $R_{remain}$
10     end
11 end
12 return $P_e$
```

blocking. Assuming that the switch is equipped with eight priority queues, Glitter has access to priorities ranging from P_1 to P_7 (typically, P_0 is reserved for other control flows) in ascending order of priority importance. UEFs are exclusively assigned P_7 to prevent them from contending for queue resources and bandwidth resources with REFs.

REF Prioritizing under Memory Exhaustion Awareness: Regarding REFs, unless memory exhaustion occurs, the extended REFs' FCT will merely lead to the sustained occupation of the remaining memory, without affecting application throughput. Hence, REFs will be assigned a lower priority.

However, applications that are generating REFs and have low or quickly filling memory, even if not blocked now, show a risk of future blocking. Hence, the urgency of these REFs is higher. Conversely, for applications with relatively sufficient local memory, particularly those that generate REFs due to having their thresholds lowered by A-ETSM but still have enough local memory, the urgency of their REFs should be lower. Therefore, we have performed a fine-grained priority differentiation of the REFs to provide better support for prone-to-blocked applications. We introduce the metric *remaining memory exhausted time* ($T_{exhausted}$), which is defined as the ratio of the remaining available memory ($M_{remaining}$) to the memory growth rate (V_{app}) to represent the estimated time it would take for the remaining local memory to be depleted entirely based on the current memory usage rate. A smaller value of $T_{exhausted}$ suggests a higher priority for the REF.

To mapping $T_{exhausted}$ to a fixed priority while assessing the urgency of the current REF across the entire cluster, we adopt a strategy inspired by [23], which involves estimating the job completion time (JCT) of each application in advance by using empirical values or recording historical information within the cluster. By utilizing this data, we can construct a cumulative distribution function (CDF) to prioritize REF during application runtime, ensuring an equitable distribution of priorities. An exemplary illustration is depicted in Figure 7, where the interval [0, 78) is mapped to priority P_2, the interval [78, 213) to priority P_3, and so forth. It ultimately forms a time-to-priority map (Map_p), used as a reference during the application runtime for priority allocation.

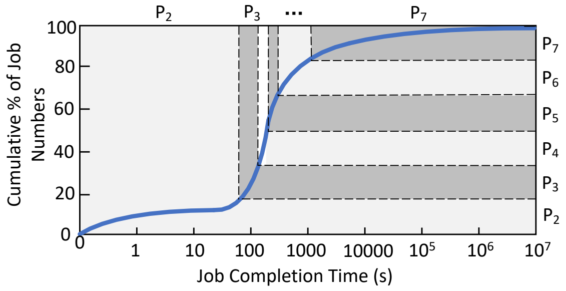


Fig. 7: The cumulative distribution function (CDF) of completion times for all applications in the cluster, which provides a equitable distribution of priorities

On the other hand, if the JCTs of applications in the cluster is difficult to estimate in advance, Glitter will consider the *remaining memory ratio* (R_{remain}) as the alternative metric for assigning priorities, which is defined as the ratio of $M_{remaining}$ to the total memory amount (M_{total}). For example, when $R_{remain} \leq 10\%$, P_2 will assigned. When $10\%<R_{remain} \leq 20\%$, priority P_3 will be assigned, and so on. The overall eviction flow prioritization assignment algorithm is displayed in Algorithm 1

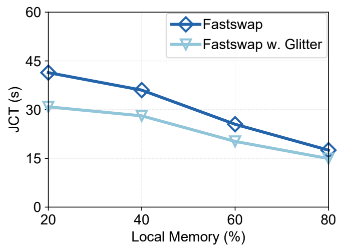
Scheduling at Switch: The switches need to enable the weighted round-robin (WRR) scheduling algorithm to mitigate the issue of low-priority REFs being starved (i.e., continuously being unscheduled), thereby protecting the performance of these applications. Fortunately, WRR, as a standard flow scheduling scheme, has been extensively employed in Mellanox commercial switches, like Dell S4048-ON.

Performance Guarantee for Non-Blocked Applications: The lower priority assigned to REFs results in a faster growth of their memory consumption. When the remaining memory allocated to non-blocked applications is fully utilized, these applications transition into a blocked state. At this point, their eviction flows are given the highest priority for scheduling, mitigating the potential degradation in their performance. To prioritize overall cluster throughput, we deem non-blocked applications' slight degradation as tolerable (no more than 1% in IV-B). Prudently, when the resulting degradation becomes intolerable, Glitter presents a safeguard mechanism to disable EFSM by assigning equal priority to all flows.

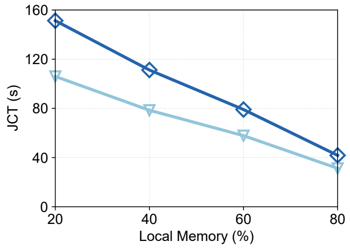
IV. Evaluation

We evaluate Glitter using NS-3 simulation to answer the following four questions:

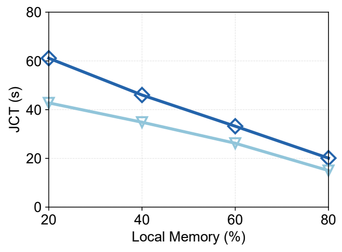
- How does Glitter perform in the large-scale scenario compared to the state-of-the-art system (§ IV-B)?
- How much do the two modules of Glitter contribute to overall performance (§ IV-C1)?
- How do the two modules of Glitter work (§ IV-C2)?



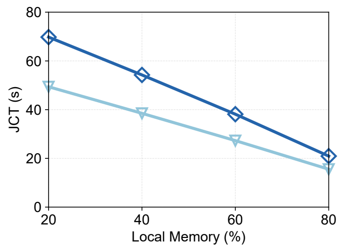
(a) Linpack



(b) Kmeans



(c) Pagerank



(d) Quicksort

Fig. 8: Comparisons of Performance between original Fastswap and Fastswap with Glitter. The lower the value, the better.

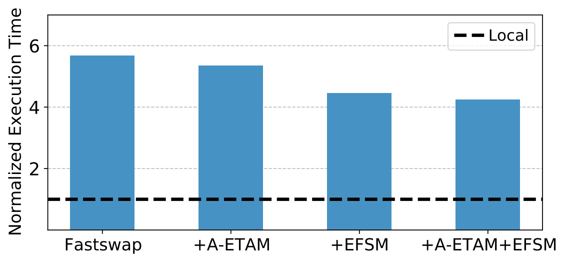


Fig. 9: Both A-ETAM and EFSM of Glitter collectively brings throughput improvements to Fastswap. Results are measured in linpak under 40% local memory.

- How does Glitter perform in various applications and under different network bandwidth pressures (§ IV-C3)?

A. Experimental Settings

Parameter Setting: Limited by experimental scale, we conducted a NS-3 simulation to evaluate the performance of Glitter in a large-scale scenario. We use a leaf-spine topology comprising two spines, six leaves, and ninety-six servers (sixteen under each leaf switch), with each server running five application tasks based on [24] (40% of machines co-run at least five jobs). The capacity of each link between spine and leaf switches is 25Gbps, and each server has a 10Gbps NIC connected to a leaf switch. All links have a $3\mu s$ propagation delay. Thirty-two servers are running memory-intensive data analytics workloads, which generate frequent application blocking during runtime. At the same time, sixty-four servers are running non-memory-intensive applications whose eviction rate surpasses the average memory growth rate, generating little application blocking.

Baseline: We incorporated Glitter into Fastswap, a state-of-the-art kernel-integrated, page-based disaggregated memory system, and compared it against the original version of Fastswap. Fastswap utilizes a fixed eviction threshold, established as the maximum memory capacity minus 8MB. Through asynchronous offloading, Fastswap delegates eviction operations to a dedicated eviction CPU. Based on a real-world test conducted under Fastswap, we set the maximum single-core eviction throughput to 287MB/s.

Workloads: We assess the performance of Fastswap with Glitter and original Fastswap using four popular workloads: `linpack`, `kmeans`, `pagerank`, and `quicksort` provided by CFM [22]. To obtain accurate simulation, we capture the memory access throughput traces using the Performance Counter Monitor provided by Intel [25] while running these workloads on an individual physical server. These recorded trace files are subsequently employed to configure the four simulated workloads. The working set sizes for each workload are 1.56GB, 4.73GB, 2.3GB, 2.4GB, respectively.

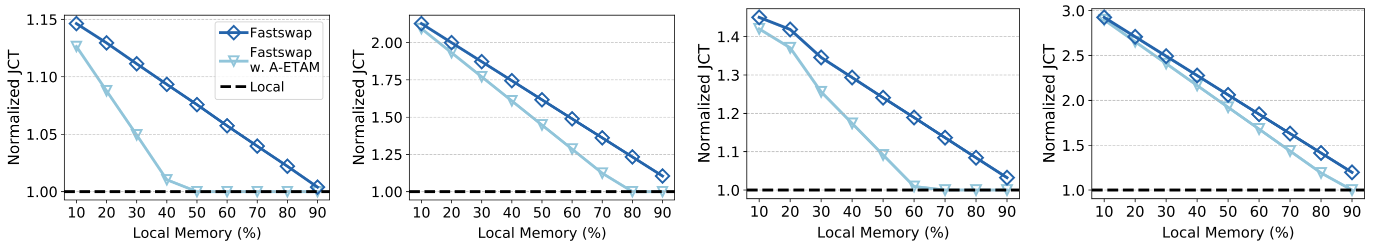
Metrics: We use Job Completion Time (JCT) as the main performance evaluation metric. Besides, we also use the number of UEFs and Flow Completion Time (FCT) to verify the effectiveness of Glitter's A-ETAM and EFSM, respectively.

B. End-to-end Performance

Figure 8 depicts an end-to-end performance comparison between the original Fastswap and Fastswap with Glitter in the large-scale scenario. As observed, the completion time of the application increases with a decrease in the proportion of local memory. This is due to the need for more remote memory use when local memory is limited, which increases communication with the remote server for data access and eviction. As a result, using remote memory adds considerable network overhead and extends the time for each data access operation. Furthermore, the limited availability of local memory results in more frequent application blocking, further impeding application throughput.

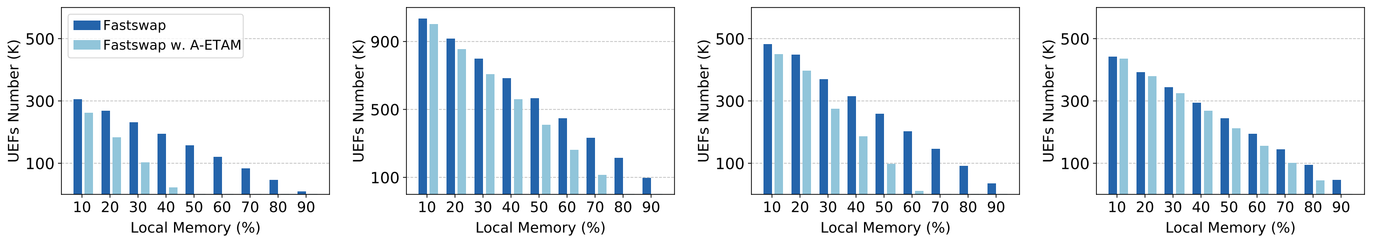
Fastswap enhanced by Glitter consistently achieves a better performance than the original Fastswap. Specifically, for the four workloads, all maximum optimizations in throughput are achieved under 20% local memory, with average improvements of 1.34×. This is because application blocking occurs more frequently as the local memory ratio decreases, leading to an increased number of UEFs and exacerbated network congestion. Benefiting from Glitter's EFSM, eviction flows generated by blocked and prone-to-blocked applications currently or subsequently impacting application blocking time will be prioritized for scheduling at the switch. In this way, Glitter effectively mitigates the effect of application blocking on throughput. Moreover, Glitter's A-ETAM reduces the occurrence of application blocking by optimizing the timing of data evictions, also contributing to throughput improvements.

Because EFSM reduces the scheduling priority of eviction flows generated by other non-memory-intensive applications, which potentially affects their performance, we conduct an



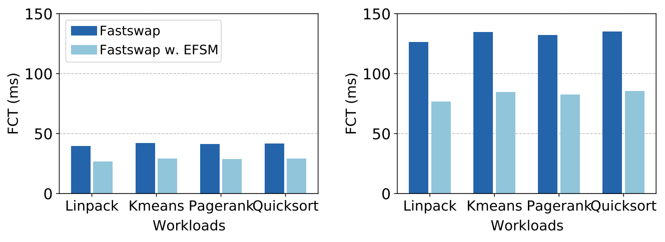
(a) Linpack (b) Kmeans (c) Pagerank (d) Quicksort

Fig. 10: Comparisons of performance between original Fastswap and Fastswap with A-ETAM in one-to-one experiment. JCT is normalized with respect to the time it takes to run the entire dataset in local memory. The lower the value, the better.



(a) Linpack (b) Kmeans (c) Pagerank (d) Quicksort

Fig. 11: Comparision of the number of UEFs between original Fastswap and Fastswap with A-ETAM in one-to-one experiment.



(a) Avg. FCT (b) 99th tail FCT

Fig. 12: Average and 99th tail FCT of UE UEFs comparisons between original Fastswap and Fastswap with EFSM in the large-scale experiment. Results are measured under 40%.

analysis to assess the impact on such applications. By measuring their JCT before and after implementing EFSM, we find that the throughput degradation for non-memory-intensive applications does not exceed 1%, which is negligible.

C. Glitter Deep Dive

1) Breaking Down End-to-end Speedup: We further explore Glitter's two modules' individual contribution to performance improvement. We use linpack as the workload for analysis. Fig. 9 breaks down the performance improvements.

As observed, utilizing A-ETAM alone gives a 6.06% throughput boost to the original Fastswap. This happens because the eviction timing in Fastswap is static. When memory growth has a tendency to potentially run out of memory, the eviction CPU in Fastswap stays idle, and by the time it begins to evict, it is already too late to prevent application blocking. Instead, Glitter's A-ETAM can adaptively adjust the timing of eviction by sensing the memory growth and the fluctuation of eviction rate, thus evicting early to prevent future blocking.

Moving forward, individual EFSM brings a 27.32% throughput improvement to Fastswap. EFSM distinguishes the importance of eviction flows based on their impact on application blocking and prioritizes the more urgent flows. On the other hand, Fastswap lacks any network optimization strategies, leading to extended completion times for UEFs and ultimately disrupting application throughput.

Finally, by employing the A-ETAM on top of the EFSM, it yields an additional 6.82% throughput improvement.

2) Effectiveness of Individual Module: In this section, we experimentally validate the effectiveness of Glitter's two modules from the perspective of algorithm design.

Effectiveness of A-ETAM: Glitter's A-ETAM dynamically adjusts the eviction threshold to optimize the timing of evicting data, thereby minimizing or even eliminating the number of application blocking occurrences. To show this effect, we test with a single compute server directly linked to a single memory server. Fig. 10 compares the normalized JCT between the original Fastswap and Fastswap with A-ETAM. As local memory grows, both systems' JCTs gradually approach the time needed to process the full dataset locally. Thanks to A-ETAM, Fastswap with A-EATM always reaches the local JCT value earlier, as early as 50% of local memory. The original Fastswap always requires the local memory occupancy above 90% to reach the same value. For kmeans, the JCT of Fastswap with A-ETAM reaches local value with 80% memory usage, while at this point, the JCT of the original Fastswap remains 1.23× higher. Additionally, for linpack, pagerank, and quicksort, Glitter achieves the

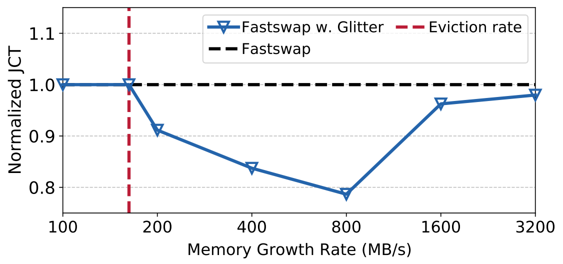


Fig. 13: Application's normalized JCT varies with memory growth rate. The lower the value, the better.

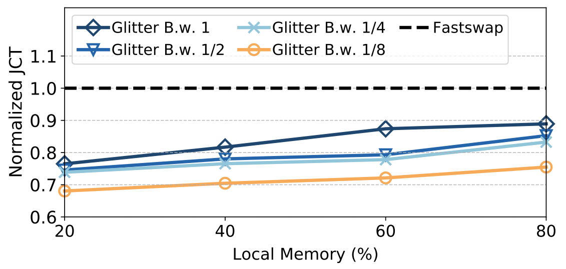


Fig. 14: Application's normalized JCT varies with the level of network congestion in linpack. The lower the value, the better.

highest performance improvements under 40%, 60%, and 90% local memory, with the improvements of 8.22%, 17.80%, and 19.73%, respectively.

We record the number of UEFs generated during the application runtime as shown in Fig. 11, which reflects the number of application blocking occurrences. A-ETAM reduces the number of UEFs across all local memory ratios, and the extent of reduction also increases as the usage of local memory increases, eventually reaching zero. Besides, we find that the degree of performance improvement brought by A-ETAM in Fig. 10 corresponds to the proportional degree of the number of UEFs reduced in Fig. 11. When the JCT reaches the local value, the number of UEFs is reduced accordingly to zero.

Effectiveness of EFSM: In scenarios where the availability of local memory is low, A-ETAM may not be able to eliminate application blocking. For instance, when the local memory ratio drops below 40%, all applications tend to generate UEFs regardless of the efforts made by A-ETAM. Consequently, it is necessary to mitigate further the impact of application blocking that has yet to be reduced through Glitter's EFSM.

Glitter's EFSM accelerates blocked applications by prioritizing the scheduling of UEFs, reducing the duration of each application blocking event. To verify it, we conduct measurements on the FCT of UEFs generated by different workloads in the large-scale scenario, comparing the original Fastswap with the enhanced Fastswap incorporating the EFSM. As shown in Fig. 12, for the four workloads, Glitter respectively achieves reductions of 30.62%, 32.59%, 30.18%, and 31.04% in average FCT, as well as reductions of 37.11%, 39.25%, 36.93%, and 37.49% in 99th tail FCT.

3) Adaptability to Different Workloads and Network Congestion: Previous experiments are constrained by limited workloads and a fixed network bandwidth. To further explore the adaptability of Glitter, we conduct two experiments to consider these two factors specifically.

Different Memory Growth Rates for A-ETAM: As a host-side factor, memory growth rate (V_{app}) for different applications affects the effectiveness of A-ETAM and has a minimal impact on the EFSM (as can be seen in Fig. 12). Therefore, our experiments are set in a one-on-one scenario to explore the performance of A-ETAM in relation to different memory growth rates. The results are measured under 40% local memory with a 16G working set size. Due to the absence of network congestion interference, the eviction rate of the eviction CPU (V_{evict}) remains constant at 287MB/s.

Fig. 13 shows that with A-ETAM, Fastswap's JCT initially remains unchanged, then drops, and finally rises as V_{app} increases, compared to the original version. This is because when V_{app} is below V_{evict}, memory will not exceed the eviction threshold. At this point, with enough local memory, applications will not get blocked and A-ETAM does not come into effect. When V_{app} exceeds V_{evict}, the eviction threshold is lowered by A-ETAM, triggering data eviction earlier. To some degree, as V_{app} increases, the benefits of A-ETAM become more evident, leading to a gradual increase in Fastswap's throughput. While as V_{app} keeps growing, the actual memory usage rate (V_{app}-V_{evict}) also increases, reducing the time to reach the eviction threshold for both versions of Fastswap, limiting A-ETAM's optimization gains.

Different Network Congestion for EFSM: To explore how varying levels of network congestion impact EFSM performance, we use the same large-scale setup as before and restrict the network bandwidth to different ratios like 1, 1/2, 1/4, and 1/8 to simulate different congestion levels. We use linpack as the test workload and disable A-ETAM. The results are shown in Fig. 14. As the available network bandwidth decreases from 1 to 1/8, the congestion within the network intensifies, leading to increased eviction flow competition. This, in turn, results in extended flow queuing delays and longer blocking periods for blocked applications. With regards to EFSM, it achieves better acceleration for Fastswap by prioritizing the scheduling of more urgent eviction flows, thereby reducing the blocking time for these applications in the case of worsening congestion. As we can see, with 20% local memory and 1/8 of the bandwidth restriction, EFSM reduces JCT by 32%. While with no bandwidth restriction, JCT is reduced by only 24%.

V. Related Work

Kernel-level Memory Disaggregation: Most existing efforts are based on the swap subsystem in the kernel to provide applications with a universal and transparent memory disaggregation platform. Infiniswap [6] designed a decentralized remote memory paging solution, which has catalyzed some subsequent research efforts. LegoOS [8] introduced the splitkernel model, dispersing traditional OS functionalities into loosely-coupled monitors to operate and manage hardware. Fastswap [1] optimizes data exchange along the critical path by specializing in RDMA queues. Canvas [10] considered multiple applications running on disaggregated memory by redesigning the swap system to fully isolate the swap path for them. Dilos [5] revamp the page fault handler to eliminate the swap cache for better performance. Hermit [4] reduced overhead on critical paths through asynchrony, batching, and eviction of CPU allocation. Glitter considers the networks and provides all of them with a more efficient eviction solution.

User-level Memory Disaggregation: To reduce data access overhead in kernel-level memory disaggregation, some efforts have implemented remote memory data management at the user-level and exposed interfaces to developers for better performance. FaRM [26] provides transaction API to allow KV-store applications to manage objects. AIFM [2] offers developers an experience similar to the C++ standard library for using remote memory. These two solutions reduce access overhead by bypassing the kernel and provide finer-grained data access to avoid I/O amplification. While Mira [13] automatically infers program behavior through program analysis, gaining the benefits of object granularity while providing transparency. Glitter's A-ETAM still applies to these works and EFSM securely preserves the prioritization of the UEFs among various sized flows for object-based systems.

VI. Conclusion

This paper presents Glitter, an adaptive, multi-level awareness eviction solution aiming to minimize the overhead of application blocking. With the joint efforts of both end-side and network-side modules, Glitter can effectively mitigate or even avoid the occurrence of application blocking and reduce the duration of each application blocking event. Through comprehensive experiments, Fastswap with Glitter achieves an average of 1.4× throughput improvement in large-scale scenarios compared to the original version.

References

[1] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can far memory improve job throughput?" in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[2] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "Aifm: High-performance, application-integrated far memory," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, 2020, pp. 315–332.

[3] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 79–92.

[4] Y. Qiao, C. Wang, Z. Ruan, A. Belay, Q. Lu, Y. Zhang, M. Kim, and G. H. Xu, "Hermit:{Low-Latency},{High-Throughput}, and transparent remote memory via {Feedback-Directed} asynchrony," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 181–198.

[5] W. Yoon, J. Ok, J. Oh, S. Moon, and Y. Kwon, "Dilos: Do not trade compatibility for performance in memory disaggregation," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 266–282.

[6] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap." in *NSDI*, 2017, pp. 649–667.

[7] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu, "Semeru: A memory-disaggregated managed runtime," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, 2020, pp. 261–280.

[8] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "Legoos: A disseminated, distributed {OS} for hardware resource disaggregation," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 69–87.

[9] C. Wang, H. Ma, S. Liu, Y. Qiao, J. Eyolfson, C. Navasca, S. Lu, and G. H. Xu, "{MemLiner}: Lining up tracing and application for a {Far-Memory-Friendly} runtime," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 35–53.

[10] C. Wang, Y. Qiao, H. Ma, S. Liu, W. Chen, R. Netravali, M. Kim, and G. H. Xu, "Canvas: Isolated and adaptive swapping for {Multi-Applications} on remote memory," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 161–179.

[11] H. Al Maruf and M. Chowdhury, "Effectively prefetching remote memory with leap," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 843–857.

[12] "The machine: the future of computing — hp." Accessd: Nov.29, 2023. [Online]. Available: https://www.youtube.com/watch?v=e3LgLA0cj50

[13] Z. Guo, Z. He, and Y. Zhang, "Mira: A program-behavior-guided far memory system," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 692–708.

[14] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "{TensorFlow}: a system for {Large-Scale} machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

[16] M. Ahmed, R. Seraj, and S. M. S. Islam, "The k-means algorithm: A comprehensive survey and performance evaluation," *Electronics*, vol. 9, no. 8, p. 1295, 2020.

[17] P. Berkhin, "A survey on pagerank computing," *Internet mathematics*, vol. 2, no. 1, pp. 73–120, 2005.

[18] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 249–264.

[19] J. Shen, P. Zuo, X. Luo, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu, "{FUSEE}: A fully {Memory-Disaggregated}{Key-Value} store," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023, pp. 81–98.

[20] S. Cui, L. Jin, K. Nguyen, and C. Wang, "Semswap: Semantics-aware swapping in memory disaggregated datacenters," in *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2022, pp. 9–17.

[21] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb *et al.*, "The design and operation of {CloudLab}," in *2019 USENIX annual technical conference (USENIX ATC 19)*, 2019, pp. 1–14.

[22] "Cluster far mem, framework to execute single job and multi job experiments using fastswap," Accessd: Feb.16, 2024. [Online]. Available: https://github.com/clusterfarmem/cfm

[23] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 221–235.

[24] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi2: Cpu performance isolation for shared compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 379–391.

[25] "Intel® performance counter monitor (intel® pcm)," Accessd: Jan.3, 2024. [Online]. Available: https://github.com/intel/pcm

[26] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "{FaRM}: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 401–414.